Zusammenfassung MOBLAB

Mobile Programming Lab

Maurin D. Thalmann

17. Januar 2020

Inhaltsverzeichnis

1 Tech-Intro

1.1 Mobile Craftmanship Mindset

→ **1:1 Portierung von Desktop zu Mobile reicht nicht aus!**

- Andere Benutzereingaben möglich auf Mobile: Touch, Pinch, Drag etc.
- Integrierte Sensoren: GPS, Kamera, Gyro, NFC, Bluetooth etc.
- Neue Einsatzmöglichkeiten: kontaktlose Interaktion, location-based, augmented etc.

- Mindset der Entwickler & Designer an neue Möglichkeiten anpassen
- Anforderungen & Wünsche der Nutzer und des Markts prüfen (User-Interaction, Plattformstandards)
- Gute Entwickler kennen Plattformen, Betriebssysteme & Bibliotheken

1.2 Entwicklung mobiler Apps

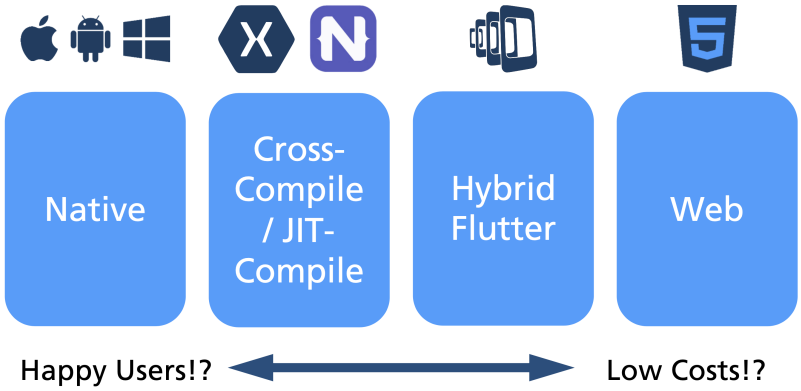

Abbildung 1: Das Spannungsfeld mobiler Entwicklung

Web-App JS, HTML, CSS mit Responsive Design, im Browser ausgeführt

Hybrid Web-App in nativem Wrapper verpackt, mit Connector-Plugins, kann als native App installiert werden (Cordova, Flutter etc.)

Cross-Compiled In Sprache X geschriebene App, wird nach Java/Object-C oder binäres Format kompiliert und somit „native“ (Xamarin, Ruby etc.)

JIT-compiled / VM Javascript-App, läuft auf JS-Engine des Zielsystems und wird dort „just in time“ kompiliert. Natives GUI und Konnektoren, um mit nativer Platform zu interagieren (NativeScript)

Native App Spezifisch pro OS programmiert, nutzt volles Featureset der Plattform, auch neuste Features

Eigenschaft	Web-App	Hybride Web-App	Cross-Platform	Native App
Plattformabh.	Nein (aber: Browser abhängig)	Nein (Grundsätzlich)	Nein (Grundsätzlich)	Ja
Entwicklungs-aufwand	1 App total	1 App + Anpassungen pro Plattform	1 App + Anpassungen pro Plattform	1 App pro Plattform
Nutzung aller nativen Möglich-keiten (OS & HW)	Nein (eher weniger als bei hybrid)	Nein (eher mehr als bei Web-App)	Jein (grundsätzlich alles möglich, aber nicht out-of-the-Box)	Ja
Vermarktung	Werbung, Abo	App-Store, Werbung, Abo	App-Store, Werbung, Abo	App-Store, Werbung, Abo
Vertrieb	Mobiler Browser	App-Store	App-Store	App-Store
App im Store / Installierbar	Nein (Aber App-Icon auf Homescreen möglich)	Ja	Ja	Ja
Performanz	Eher schlecht	Je nach Ansatz, eher schlechter als nativ	Gut	Gut

Abbildung 2: Übersicht mobiler Entwicklungsansätze

Welches ist jedoch der beste Ansatz für eine App? Immer abhängig von Anforderungen und Möglichkeiten. Mögliche Szenarien:

- Info-App ohne viel Interaktion → **Web-App**
- 100% natives Look n Feel → **Nativ**
- Viele Plattformen nativ unterstützen, preiswert, .NET oder Angular Knowhow vorhande → **Cross-/JIT-Compile (Xamarin, NativeScript)**
- Gemeinsame Codebase, Crossplattform, eigene Widgets, Hot-Reload, kleine App / Prototyp → **Flutter**
- HW-Features benötigt auf verschiedenen Plattformen (NFC, Kamera, BT, Storage...) → **Hybrid / Nativ**

1.3 Android Grundlagen

- Android-Applikationen bestehen aus vier Komponenten:
  Activity UI-Komponente, entspricht typischerweise einem Bildschirm
  Service Komponente ohne UI, Dienst läuft typischerweise im Hintergrund
  Broadcast Receiver „Event-Handler", reagiert auf Broadcastsnachrichten (Intents)
  Content Provider Komponente, ermöglicht Datenaustausch zwischen versch. Applikationen
- **Android Runtime (ART)** verwaltet die einzelnen Komponenten einer Applikation
  - Mit Intent-Mechanismus kann eine Komponente eine andere Komponente aufrufen
  - Komponenten müssen beim System registriert sein (teilweise Rechte = Privileges)
  - System verwaltet Lebenszyklus von Komponenten (Gestartet, Pausiert, Aktiv, Gestoppt etc.)
- Android empfiehlt für Hintergrundaufgaben nicht mehr Services, sondern `android.app.job.JobScheduler`
  Neu ist JobScheduler auch in WorkManager von Android Jetpack integriert

1.3.1 Android Manifest

- Beschreibt statische Eigenschaften einer Applikation
  - Basis Java-Package-Name
  - Benötigte Rechte (Internet, Kontakte etc.)
  - Deklaration von Komponenten
    - Activities, Services, Content Providers, Broadcast Receivers
    - Name (+ Basis-Package = Java-Klasse)
    - Anforderungen für Aufruf (Intent-Filter) für **A, S, BR**
    - Format der gelieferten Daten für **CP**
- Diese Infos werden bei der App-Installation im System registriert
- Zusatzinfos (Version, ID etc.) befinden sich im Build-Skript (da diese build-abhängig sein können)

1.3.2 Android Projekt-Struktur

- Manifest
- Java-Code: Activities (App-Logik, Tests usw.)
- Ressourcen (`res`)
  - Bilder (`drawable`)
  - Layouts (`layout`)
  - Menus (`menu`)
  - Werte (`value`)
- Gradle Skripts (Angaben zum Build)

1.4 Android Jetpack & App-Architektur

Zwei massive Neuerungen in letzter Zeit:
Seit 2019: Kotlin wird primäre Android-Sprache
Seit 2018: Android Jetpack wird ins Leben gerufen

1.4.1 Android Jetpack

- Sammlung von SW-Komponenten, die bei der Entwicklung von state-of-the-art Android-Applikationen unterstützen soll
- Jetpack-Komponenten im `androidx.`-Namespace, wurden teils aus Standard-API hierhin verschoben
- Alle Komponenten sind rückwärtskompatibel, können unabhängig von Android-Release-Zyklus aktualisiert und verwendet werden
- Jetpack wird von Google entwickelt und dokumentiert

Jetpack wird unterteilt in 4 Bereiche:

- Architecture
  - Data Binding, Lifecycles, LiveData etc.
- UI
  - Animation/Transitions, Auto, TV, Wear, Emoji, Fragment etc.
- Behavior
  - Download Manager, Media & Playback, Permissions, Notifications etc.
- Foundation
  - AppCompat, Android KTX, Multidex, Test

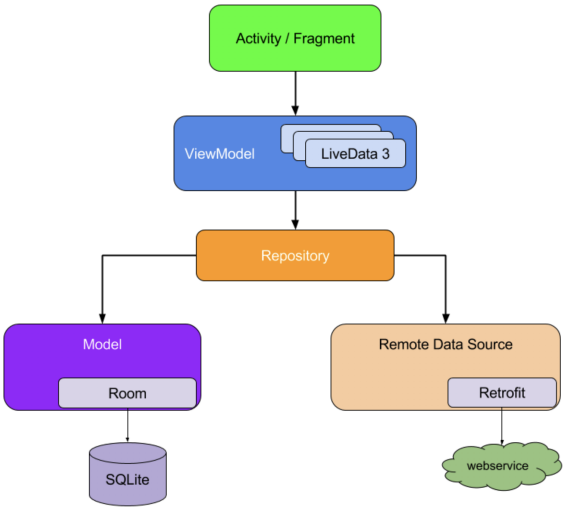

Abbildung 3: Empfohlene App-Architektur

1.4.2 Android Architecture Components (AAC)

- Android Architecture Components enthalten eine Reihe von Lifecycle-bewussten Komponenten
- Komponenten helfen bei der Lösung von Problemen mit Konfigurationswechsel, Persistenz, Memory-Leaks und asynchronem Datenupdate auf dem UI
- AAC definieren seit 2017 eine standardisierte Vorgehensweise und stellen die offiziell empfohlene Lösung von Google dar

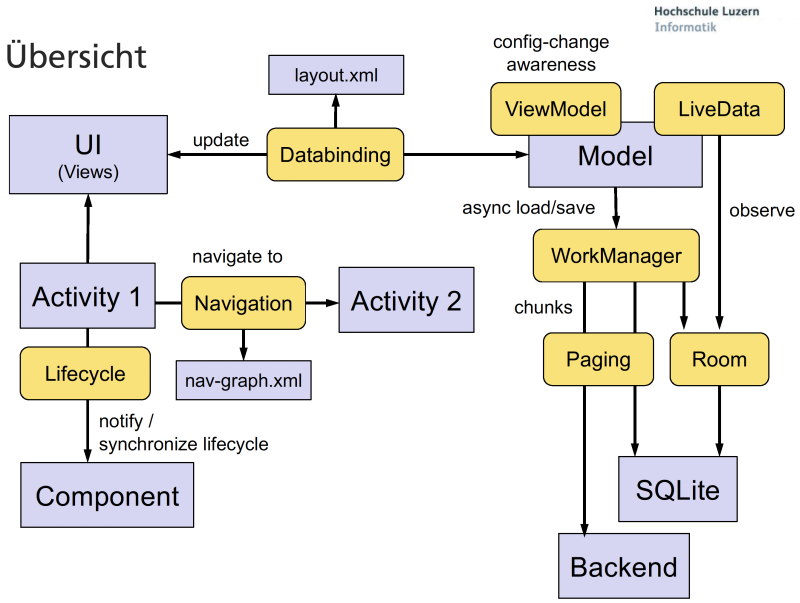

Abbildung 4: Übersicht der Android Architecture Components

Data Binding

- Separiert das UI von den Daten
- Synchronisiert UI mit Daten (1-way oder 2-way Binding)
- Verwendet „Binding Expressions“ mit @... Syntax im Layout-File, um View-Attribute zu initialisieren

Lifecycle

- Kann Code aus den Lifecycle-Hooks von Activities entfernen und direkt auf der beobachtenden Komponente implementieren.
- `Lifecycle` ist ein Objekt, welches den Lebenszyklus einer Komponente abbildet (Activity, Fragment etc.)
- Andere Komponenten können das Lifecycle-Objekt beobachten und auf Lifecycle-Events reagieren

LiveData

- Updates können im Hintergrund erfolgen, werden aber nur ausgeführt, wenn der Observer der LiveData in einem aktiven Zustand ist (`STARTED, RESUMED`)
- `LiveData` ist eine lifecycle-aware Observable-Klasse
- Kann als Source für Data Binding verwendet werden, um Aktualisierungen des UI während der Laufzeit zu forcieren

Navigation

- Android gibt neu Navigationsprinzipien vor, hierbei hilft die `Navigation`-Komponente bei der Implimentierung dieser Prinzipien
- `Navigation` basiert Navigation-Graph (Resources) mit Destinations (Knoten) und Actions (Kanten) *(Navigation-Graph wird von Hand oder mit Navigation-Editor in Android Studio erstellt)*

- Navigation benötigt `NavHostFragment` im Layout (um die Zielfragmente einzublenden); Aus dem Code heraus wird mit einer `NavController`-Instanz navigiert

Paging
- Es müssen nicht alle Daten auf einmal geladen werden: schneller und weniger Load
- Unterstützt asynchrones Laden von Daten
- `PagedList` und `PagedListAdapter`, um bei Bedarf weitere Daten in einer `RecyclerView` zu laden

Room
- Bessere Abstraktion, Speichern/Laden von Modellobjekten, kein handgestricktes SQL-Mapping
- Room ist ein ORM (Object-Relational Mapper) für SQLite
- Arbeit mit Entities (Modell-Objekten) und DAO-Pattern anstatt SQL

ViewModel
- Weniger Aufwand für Behandlung von Konfigurationsänderungen (keine Serialisierung nötig)
- Kapselt UI-Daten so, dass sie bei Konfigurationsänderung einer Activity in-memory erhalten bleiben
- Aber: Für den Fall eines App-Kills durch das OS müssen Daten immer noch persistiert werden!

WorkManager
- Kein Kopfzerbrechen über Regeln für Hintergrundtasks, Synchronisation von UI via LiveData
- `WorkManager` führt asynchrone `WorkRequests` sofort oder zu geeignetem Zeitpunkt aus
- Respektiert Doze-Mode, versucht Ressourcen zu sparen und Load zu minimieren. Je nach Zustand von App/System werden Tasks unterschiedlich scheduled.
- Pro WorkRequest wird ein LiveData-Objekt erzeugt, Zustand und Daten sind darüber beobachtbar

App-Architektur: Tipps & Empfehlungen
- Standards / Patterns soweit möglich benutzen
- Aber: Kein Over-Engineering!
- Alle AAC können einzeln oder zusammen verwendet werden
- Herausfinden, was im Projekt am besten funktioniert bzw. am meisten Sinn macht
- Vorsicht bei Background-Tasks
  - System zunehmend restriktiver, grosse Änderungen mit API 26
  - „Background = Service“ gilt nicht mehr
  - WorkManager meist die bessere Alternative, Service für Logikexport

2 SA - Kotlin & Android

2.1 Sprachübersicht

- Variablen zwingend mit `var` (veränderbar) oder `val` (unveränderbar) kennzeichnen
- Primitive Datentypen gibt es nicht, dafür Klassen für diese (Int, Double etc.)
- Kontrollstruktur-Ausdrücke haben immer einen Wert
- Keine checked Exceptions
- Strichpunkt am Ende einer Zeile ist optional
- Typinferenzen, weniger explizite Typangaben als bei Java
- Nullable-Typen können gegen NullPointerExceptions vorbeugen
- Mit Extensions Klassen ohne Vererbung erweitern
- Data Classes für kompakte Klassendeklaration inkl. automatisch implementierten equal / toString / hashCode Methoden

2.2 Spracheigenschaften (eine Auswahl)

Variablen-Deklaration: var, val & Typinferenz

- `var` (veränderbar) und `val` (unveränderbar)
- Typangaben können grundsätzlich weggelassen werden
  (Compiler erkennt Datentyp automatisch und weist diesen korrekt zu)
- Java kennt für lokale Variablen seit 2018 ähnliche var-Syntax mit Typinferenz für lokale Variablen
- Bei Kotlin: Typangabe nach dem Namen
  Bei Java: Typangabe vor dem Namen

Nullable-Typen, Safe-Calls & Elvis-Operator

- In Kotlin können jegliche Typen nie den Wert null annehmen
  (ausser sie werden explizit als nullable deklariert)
- Variable, die Wert null zulassen soll, so deklarieren mit `[type]?`
- Nullable-Variablen können nicht direkt abgerufen werden (könnte ja null sein)
  - Variable entweder erst auf null überprüfen
  - Safe-Call Operator `?.` nutzen (druckt Wert der Variable oder null)
  - Not-Null-Assertion-Operator `!!` (wirf NullPointerException, falls Wert null ist)
- Spezialfall: falls nicht-null, nimm den Wert und sonst X
  - Elvis-Operator `?:` (folgend auf diesen kann ein Wert angegeben werden)
  - Zusammen mit Safe-Call-Operator nützlich, da so ein Alternativwert angegeben werden kann, falls eine Variable doch null sein sollte

Funktionen: Benannte Parameter & Default-Werte

- Schlüsselwort `fun` zur Deklaration von Funktionen
- Benannte Parameter und Default-Werte für Parameter
  - Benannte Parameter für bessere Lesbarkeit
  - Default-Werte für übersichtliche Aufrufe
  - Bei Aufruf einer Funktion mit bennanten Parametern können diese beliebig angeordnet werden, jedoch darf beim Aufruf nicht zwischen unbenannt und benannt gemischt werden, da sonst die Position der Parameter nicht mehr stimmen könnte
- Default-Werte helfen bei der Code-Einsparung (bei Java brucht es bei x Parametern x Methoden)
- Weitere spannende Möglichkeiten von Funktionen:
  `tailrec` für endrekursive Funktionen, Funktionen mit nur einem Ausdruck, Funktionen mit expliziten Rückgabetypen, Modifier `infix` für infix-Aufrufe (infix-Notation und ohne Klammern)

Extension: Erweiterung ohne Vererbung

- Extensions, um bestehende oder auch fremde Klassen um Funktionen zu erweitern
  z.Bsp. kann Klasse `Int` um Funktion `myPrettyPrint()` erweitert werden

2.3 Kotlin & Android

- Kotlin offizielle Androidsprache, da kompakt, ausdrucksstark, Typ- und Null-sicher
- Parallele Verwendung von Java und Kotlin möglich für Android-Entwicklung
- Aus Kotlin-Klasse kann auf Java-Klasse zugegriffen werden (und umgekehrt)
- Sicherer als Java dank non-nullable Typen und kompakteren Konstrukten
  40% weniger Zeilen Code als bei Java
- Android findViewById() entfällt bei Kotlin, da direkt Extension-Properties für alle id's in Ressourcen-Dateien angelegt werden, so kann direkt auf jede id zugegriffen werden

3 SA - Data Binding & ViewModel

3.1 Kontext

- Eine Activity ist ein Bildschirm, zu jeder Activity gehört ein UI-Layout in einem XML, welches darzustellende Views definiert
- UI-Initialisierung in der `onCreate()` einer Activity
- Code mit Zuständigkeit für Daten, UI und Eventhandling wird vermischt
- Data Binding soll Daten und UI strikt trennen
- ViewModel soll Behandlung von Konfigurationsänderungen von der Activity separieren

3.2 Data Binding

- Anbindung eines Datenmodells an UI-Komponente
- Modell für UI-Daten erzeugen und mit UI verbinden
- *One-Way-Binding*: Wird Datenmodell modifiziert, aktualisiert UI automatisch mit neuen Daten
- *Two-Way-Binding*: Auf UI eingegebene Daten werden auch mit dem Modell synchronisiert
- Eliminiert viel Listener- und Update-Code in den Activity-Klassen
- Test- und Wartbarkeit wird erhöht, im Best Case nur noch Modell-Logik testen
- Data Binding über spezielle Layout-Files definieren
  - `<layout>` mit zwei Unterelementen
    - `<data>`: deklariert Namen und Typen der verwendeten Datenquellen
    - `View`: entspricht Root-Komponente eines normalen Layout-Files
  - Für beliebige Attribute von View-Elementen Binding-Expressions definierbar
    (werden zum Zeitpunkt der Instanzierung ausgewerten und deren Resultat als Wert des Attributs gesetzt)
  - Binding-Expressions innerhalb des Markers `@...` definiert, können auf Variablen/Modellobjekte referenzieren, welche im <data> Element des Layouts definiert worden sind
  - Binding-Expressions können auch einen Default-Wert enthalten (falls sie Null sind)
- Model-Klasse implementiert definierte Properties, am besten als public-Felder
- Klassen werden automatisch generiert, welche die Binding-Logik für jedes Layout implementieren (mit Suffix *Binding* nach dem Activity-Namen)
- Initialisierung des UI mit `DataBindingUtil.setContentView()` in `onCreate()`
  Die Model-Klasse wird ebenfalls hier geladen, mit Werten belegt und gebunden
- Observables für Aktualisierungen in der Model-Klasse definieren

3.3 ViewModel

- ViewModel, um den mit UI verbundenen Zustand in-memory zu speichern
- Mit Data Binding kombinierbar, da Data Binding nach erneuter Initialisierung der Activity sonst den Wert null oder Default-Werte wieder annimmt
- ViewModel-Klasse erstellen, welche von ViewModel ableitet / erbt
- Hier Observables definieren, mit den zu überwachenden UI-Elementen
- ViewModel-Klasse muss mit der Activity verbunden werden, mit welches es seine Daten synchronisiert (`ViewModelProviders`)
- ViewModel muss einen Default Konstruktor haben, welcher eine Application als Argument nimmt, für den Fall von AndroidViewModel

- Bei Verbindung mit Backend / DB überprüfen, ob diese schon einmal geladen worden, da ansonsten das ViewModel zurückgesetzt wird wenn keine Verbindung zum Backend besteht, was unerwünscht ist

4 SA - Fastlane

5 SA - Unreal Engine

6 SA - Xamarin.Forms

7 SA - PWA: Progressive Web Apps